Exploring Light-Field Reconstruction on Realistic Camera

b00902107 Shu-Hung You

1 Introduction

In the final project, I extends *PBRT* [2] with part of the system proposed in [1] and tested it together with the realistic camera module implemented in homework 2. The system utilizes the anisotropy in the light field, and reconstructs dense pixel "samples" from the rather sparse input samples. The work is especially suitable for reconstructing defocus effects (and motion blur).

In *PBRT*, the `SamplerRenderer` renders the image by integrating over the 4D $xyuv$-hypercube, where x, y denotes coordinate on the image plane and u, v denote coordinate on the lens. The defocusing effect is then caused by multiple different intersection points contributing to a single pixel of the image.

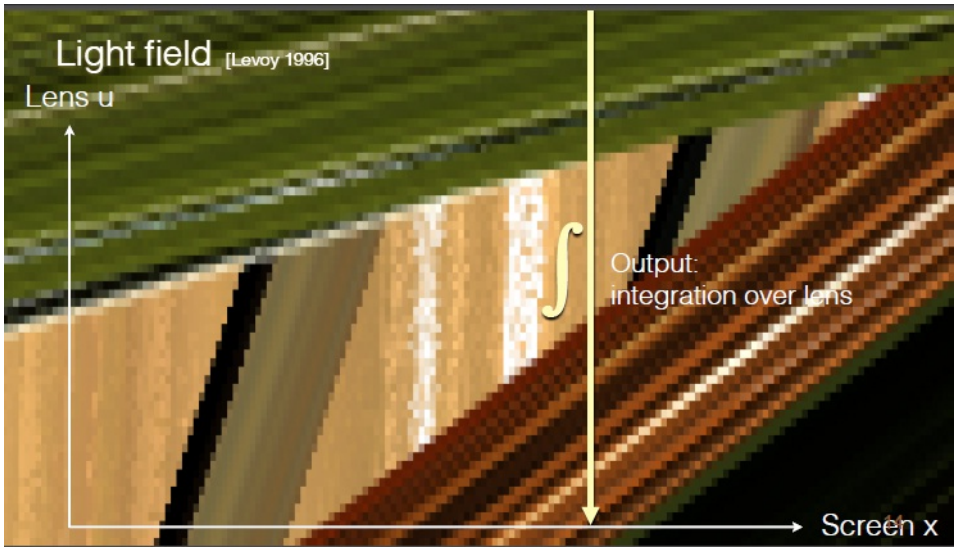


Figure 1. Demonstration of the xu-plane in [1].

The defocusing effect is depicted in Figure 1. For the simple pin-hole camera, for every xy-coordinate there exists exactly one uv-coordinate that contributes to the pixel, hence only one row in Figure 1 is effective for each y. However, for realistic cameras the out-of-focus points actually displace when viewed through different positions on the lens. This leads to the slant trajectories in Figure 1. For points on the focus, the trajectories are almost vertical.

Figure 2. The light field is anisotropic [1].

On the other hand, the light field is actually anisotropic. Given a point in the scene, its contribution to the light field is approximately constant over different point on the lens as in Figure 2. The resulting image of the xu-plane hence consisting of several different lines segments, each resulting from some fixed point in the scene. This corresponds to the intuition that the scene should move smoothly when we stroll through the lens. The usual Monte-Carlo integration method (integrating over u) does not exploit this property but instead sampling the xu-plane vertically, thus obtaining signals with large variance at out-of-focus points.

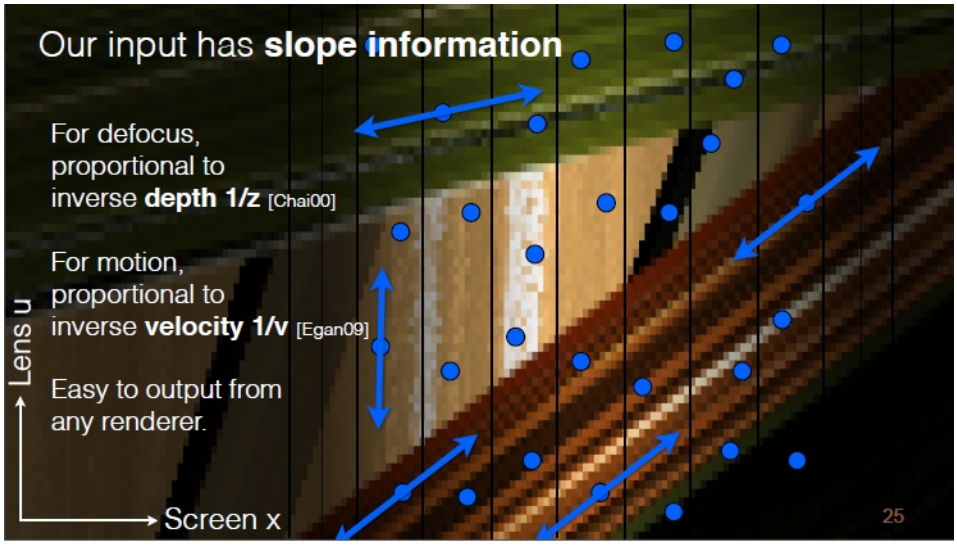


Figure 3. (Sparse) samples with slopes [1].

In the proposed system, the anisotropy of sampling points are utilized to reconstruct the light field. Given the relatively sparse input samples of the light field, the system estimates the slope of each sample by its depth with formula

$$\frac{\partial u}{\partial x} \approx \frac{C_1}{z} + C_2$$

where C_1, C_2 are predetermined constants drawn using the thin-lens approximation. Then for any reconstruction location $(\hat{x}, \hat{y})$ on the image plane and any sample position on the lens $(\hat{u}, \hat{v})$, the algorithm first moves each sample at (x, y), (u, v) along its trajectory to the new (x', y'), $(\hat{u}, \hat{v})$ location (i.e. to the same horizontal line on the xu-, yv-plane). Visible samples (x', y') near the point $(\hat{x}, \hat{y})$ are then filter together to produce the radiance of the sample $(\hat{x}, \hat{y})$, $(\hat{u}, \hat{v})$.

The reconstruction technique is claimed to be applicable to a wider range other than defocus, such as motion blur, soft shadows and refocusing of the picture. I did not repeat those experiment, though.

2 Algorithm and Implementation

The algorithm works by first render a coarse version of the image, then re-render the image using dense, reconstructed samples on the light field. Given a reconstruction location $(\hat{x}, \hat{y})$, $(\hat{u}, \hat{v})$, the reconstructed radiance $R(\hat{x}, \hat{y}, \hat{u}, \hat{v})$ is calculated by

1. Reproject samples to the new positions using $(\hat{u}, \hat{v})$. The formula is

$$x^*(u) = x + (u - \hat{u})\frac{\partial x}{\partial u} = x + (u - \hat{u})\left(\frac{C_1}{z} + C_2\right)$$
$$y^*(v) = y + (v - \hat{v})\frac{\partial y}{\partial v} = y + (v - \hat{v})\left(\frac{C_1}{z} + C_2\right)$$

2. Determine which (set of) reprojected samples are visible at the reconstruction location. The reprojected samples near $(\hat{x}, \hat{y})$ with non-crossing trajectories are grouped together to form several surfaces. The samples in the up front surface are the visible ones.

3. The visible samples are filtered together to form the reconstructed radiance.

The reconstructed samples are then filtered through the pixel filter, P, to produce the actual radiance at coordinate $(\hat{x}, \hat{y})$.

$$L(x, y) = \frac{1}{N}\sum_i R(x_i, y_i, u_i, v_i)P(x_i, x_j)$$

It is possible to accelerate the algorithm by building a variant of bounding-volume hierarchy (BVH) data structure to aid the search of close reproject samples in step 2. I use a simple heuristic though, at the cost of lowering the quality of the image since defocussed points could move a long distance in the rendered image.

My implementation of the reconstruction algorithm is modified from the *SamplerRenderer* class presented in *PBRT*. I copied the file `renderers/samplerrenderer.*` to `renderers/reconrenderer.*`, and add a new type of renderer in `core/api.cpp`.

```
Renderer *RenderOptions::MakeRenderer() const
{
        // ...
        if (RendererName == "reconstructed")
        {
          Transform *pWorld2Camera;
          transformCache.Lookup(WorldToCamera[0], &pWorld2Camera, NULL);
```

```
        renderer = new ReconRenderer(sampler, camera, surfaceIntegrator,
                                     volumeIntegrator, nsamp, pWorld2Camera);
    }
    // ...
}
```

Then in `ReconRenderer::Render(const Scene *scene)`, the renderer issued a two-stage rendering process with helper classes `ReconRendererInit` and `ReconRendererTask`. The `ReconRendererInit` performs the sampling process and radiance calculating, constructing all `ReconSample_t` of the image. The class `ReconRendererTask` calculates a dense set of reconstructed samples, and render the final image.

```
struct ReconSample_t {
  float distz, depth;
  float imageX, imageY, lensU, lensV;
  Spectrum L;
  // some other methods
}
```

The `ReconRenderer` have several parameters. (The # of reconstructed samples per pixel is set on the camera).

- `"integer pixelsamples"`, the # of input samples per pixel.
- `"integer range"`, the range of input samples to search for.
- `"float alpha"`, the α parameter of the Gaussian filter, used to filter visible samples.
- `"float weight"`, the weight of the ray (to adjust brightness).
- `"float C1"` and `"float C2"`, the constant C_1, C_2.

3 Experiment and Discussion

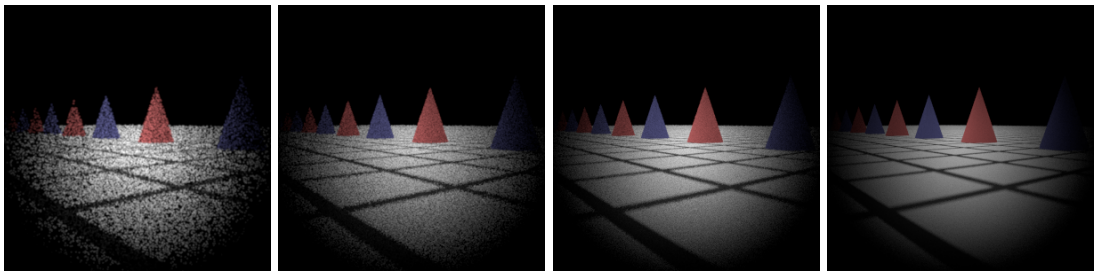


Figure 4. Reconstruction with Various Number of Input Samples. The images are rendered with 1, 4, 16 and 64 input samples from left to right, respectively

The result is not quite satisfactory, though. I have experimented rendering images in homework 2 with various number of input samples as in Figure 4. All images have 256 reconstruction locations (per pixel). Compared to the ground truth, though the image contains less fractal, the black "holes" still exist and the results are not acceptable. The result of other images in homework 2 are given in Figure 5.

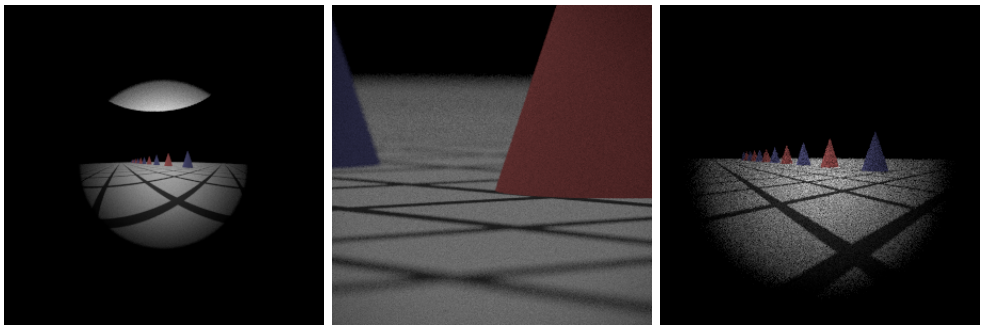
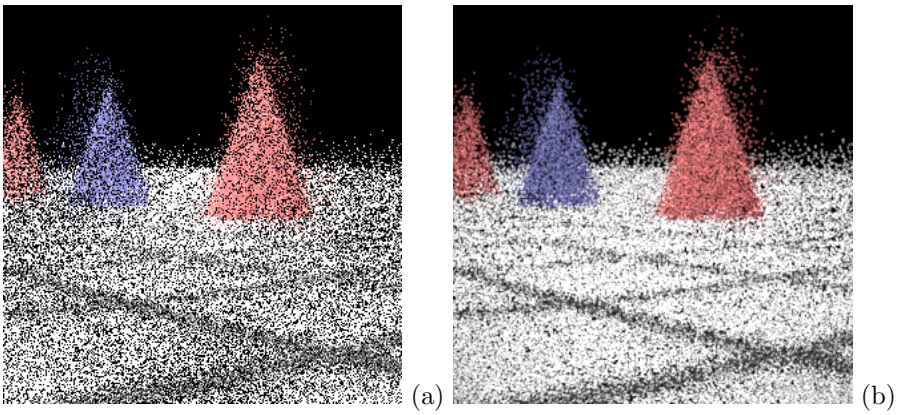


Figure 5. Reconstruction Result. All images are rendered witn 64 input samples and 256 reconstructed samples.

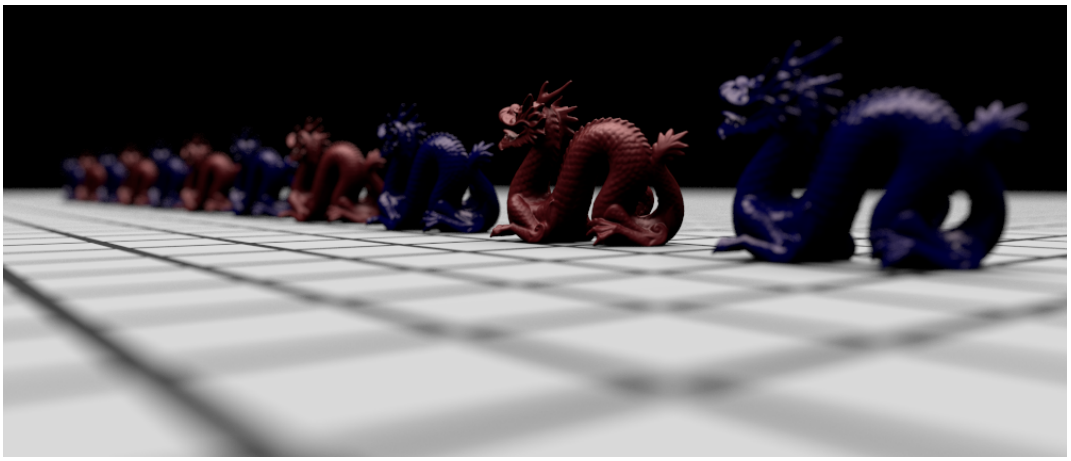
However, in some cases the image can have a relatively good result compared to its original input samples. In Figure 6, the two images are the raw input samples and the reconstructed result, respectively. There are 4 input samples per pixel and 256 reconstructed samples. Interestingly, the images grow darker as the number of input samples increases.



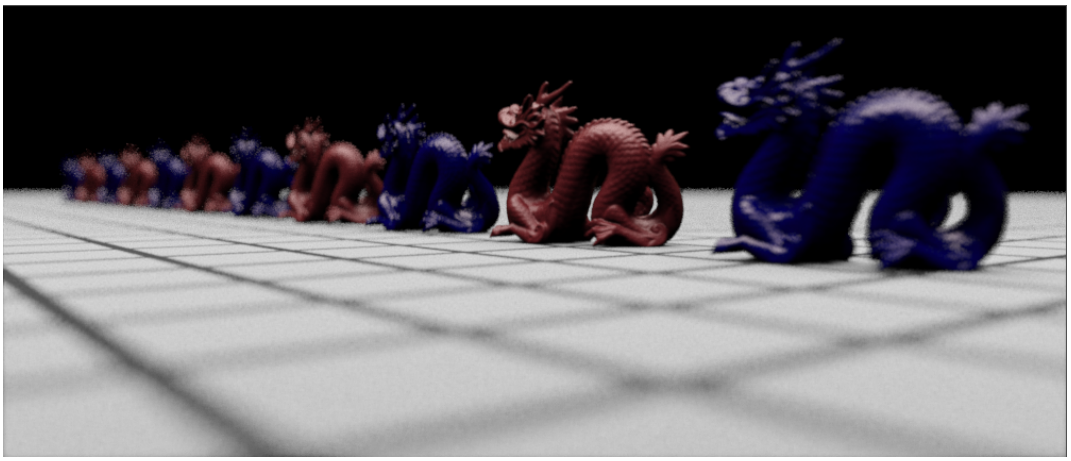
(a) (b)

Figure 6. Reconstruction from Relatively Low Resolution.

I also tried to render the realistic `dof-dragons.pbrt` taken from [3]. The result is shown in Figure 7. Though it looks better than the above ones, there are still non-negligible differences between the rendered image and the ground truth. Nevertheless, the rendering time is less than the original *PBRT*. The paper [1] has given a much compelling speed-up using BVH and GPU.



(a)



(b)

Figure 7. Experiment Result. (a) is rendered with 256 samples by `SamplerRenderer` in 9403.1s (b) is reconstructed from 16 samples in 4066.8s.

4 References

1. Jaakko Lehtinen et al, *Temporal Light Field Reconstruction for Rendering Distribution Effects*. Proceedings of ACM SIGGRAPH, 2011.

2. Pharr, M., and Humphreys, G. 2010. *Physically Based Rendering, 2nd ed.* Morgan Kauffmann.

3. *Scenes for PBRT*. `http://www.pbrt.org/scenes.php`

Appendix A. Environment Setup and Compilation

My code compiles both on Windows (with MinGW) and Linux (CSIE Workstation). However, the `Makefile` and several codes have been patched in order to enable C++11 features. Please see the accompanying patch. To compile the code, just go to the root directory of the source files and type `make pbrt`.